A Power-Efficient Methodology for Mapping Applications on Multi-Processor System-On-Chip Architectures

Giovanni Beltrame, Donatella Sciuto, and Cristina Silvano

Politecnico di Milano, DEI, Milano 20133, Italy

Abstract. This work introduces an application mapping methodology and case study for multi-processor on-chip architectures. Starting from the description of an application in standard sequential code (e.g. in C), first the application is profiled, parallelized when possible, then its components are moved to hardware implementation when necessary to satisfy performance and power constraints. After mapping, with the use of hardware objects to handle concurrency, the application power consumption can be further optimized by a task-based scheduler for the remaining software part, without the need for operating system support. The key contributions of this work are: a methodology for high-level hardware/software partitioning that allows the designer to use the same code for both hardware and software models for simulation, providing nevertheless preliminary estimations for timing and power consumption; and a task-based scheduling algorithm that does not require operating system support. The methodology has been applied to the co-exploration of an industrial case study: an MPEG4 VGA real-time encoder.

1 Introduction

Technological advances have made multiprocessor implementations of embedded systems a viable alternative to traditional single-processor and pure-hardware designs. Such multiprocessor designs offer high levels of performance, flexibility and, at the same time, promise low-cost and power-efficient implementations. One of the most promising approaches to design such systems is the Multiprocessor System-on-a-Chip (MPSoC) paradigm. A typical MPSoC system consists of a number of processing elements (PEs), which can be programmable processors or fixed application-specific coprocessors, and storage elements (SEs) connected to PEs via an on-chip communication architecture. As a result, MPSoC architectures represent heterogeneous systems that offer flexible parallel processing resources for implementation of bandwidth-demanding multimedia applications.

However, MPSoC platforms introduce several design challenges associated with their parallel and heterogeneous architecture. Platform-based design [1] faces the problem of defining a configurable microarchitecture platform, onto which an application can be mapped through a well defined parallel programming model, specified via an application-program interface (API) platform. The mapping of an application to a microarchitecture platform starts from a complex system specification and goes through a possible extensive design space exploration phase. In this context, the reuse of a large

Please use the following format when citing this chapter:

Beltrame, G., Sciuto, D. and Silvano, C., 2007, in IFIP International Federation for Information Processing, Volume 249, VLSI-SoC: Research Trends in VLSI and Systems on Chip, eds. De Micheli, G., Mir, S., Reis, R., (Boston: Springer), pp. 177–196

base of existing software to perform the exploration of different possible implementations constitutes an important concern. The existing software commonly written in C/C++ language with a single-processor architecture in mind cannot be directly reused in a multiprocessor environment, especially if it consists of a heterogeneous mix of different software and hardware components. The existing software needs to be adapted to the parallel capabilities of the architecture. Furthermore, to enable fast and flexible exploration of the possible application-to-architecture mappings, it is necessary to automate the hardware-software partitioning of the application. Therefore, there is a need for a disciplined approach based on a unified parallel modelling paradigm that would enable a smooth translation of existing sequentially-coded software algorithms into their parallel models suitable for the design space exploration of MPSoC platforms.

In this work, we show how it is possible to map an application to the MultiFlex [2] platform, exploiting the features offered by the combined use of the DSOC and the SMP programming model to provide a novel way of performing initial partitioning and exploration. The use of the DSOC programming model together with the transaction-level modeling (TLM) [3] infrastructure allows easy moving of a component from hardware to software and vice-versa.

Once the application has been partitioned and mapped to the target platform according to its performance and power constraints, it is possible to further optimize its power consumption with the use of a proper Dynamic Power Management System (DPMS). In this work we extend our previous work [4] with a task-based DPMS scheduler, that optimizes power consumption without affecting the system's performance.

The rest of this paper is structured as follows: Section 2 describes current approaches to the parallel mapping problem; Section 3 outlines the proposed mapping flow; Section 4 introduces power optimization to the overall architecture; Section 5 describes an industrial case study to which the proposed methodology was applied; and finally, Section 6 draws some concluding remarks.

2 Related work

The present paper focuses on a mapping methodology of applications onto MPSoC platforms in order to identify the best trade-off of power and performance behavior of the application on a given platform configuration. The next step is task allocation for dynamic power consumption optimization. Most literature is either focused on programming models to solve the mapping of software applications onto specific platforms or on the scheduling for dynamic power management on multiprocessors. So far, no paper considered yet the entire design flow from a comprehensive perspective.

2.1 Designing MPSoCs

The architectural changes introduced in the emerging MPSoCs have a direct consequence in how software engineers program. This fact has already been acknowledged by several researchers, who have proposed preliminary solutions. Most of them agree on the importance of new high-level programmer views of SoC.

A number of programming models focused on multiprocessor SoCs have been presented, such as the MESCAL approach [1], which has served as base for further different programming models. Nevertheless, most of them are application or domain specific.

A more general approach composed of two SoC parallel programming models has been introduced in [5]. The Distributed System Object Component (DSOC) model and the Symmetric Multi-Processing (SMP) model are inspired by leading-edge approaches for large system development, but adapted and constrained for the SoC domain.

Various actor-oriented frameworks are proposed to capture arbitrary Models of Computation (MoC) for the purpose of system level modeling and tool supported paths to exploration, implementation and/or verification [6]. The modeling strategy presented in this paper can be implemented on top of any of these MoC generic frameworks. We selected SystemC mainly because of the broad user acceptance and commercial tool support. Complementary to our top-down refinement flow, the Component Based Design paradigm [7] advocates the bottom-up platform composition from a parameterizable IP library, containing off-the-shelf processing elements, communication fabrics and hardware dependent software layers. This approach is clearly advantageous for the rapid exploration and implementation of the general purpose portion of the application, whereas our approach is focused on application specific architectures executing the data-processing part.

The highest possible abstraction level for design space exploration and application mapping is static performance analysis [8, 9]. Other approaches are closer related to simulation frameworks for top-down exploration and refinement like ARTEMIS [10] and StepNP [5]. Some recent works present simulation frameworks for mapping applications based on SystemC [11], and mapping and scheduling of applications on parallel architectures [12, 13].

2.2 Power optimization

Dynamic Power Management (DPM) is a design methodology that dynamically reconfigures an electronic system to provide the requested services and performance levels with a minimum number of active components or a minimum load on such components.

Dynamic Voltage/Frequency Scaling (DVFS) requires processors to adapt their voltage and frequency at run-time, according to some control actions. The work in [14] introduce architectural and implementation issues together with energy saving bounds concerning DVFS techniques.

A significant amount of research on DVFS scheduling and algorithms have been proposed on both single and multi-processor systems. DVFS has been implemented in several contemporary microprocessors as Intel XScale, AMD Mobile K6 Plus and Transmeta Crusoe [15]. These can be classified as compile-time and run-time policies [15]. Run-time policies have drawn more attention because of the ability to reduce energy consumption in response to workload variations. A run-time DVFS policy consists of two elements [15]:

– *Scaling points*: these are the positions where voltage/frequency scaling occurs. They can be signaled by timer interrupts, cache misses, etc. The time frame enclosed by two scaling points is referred as a *scaling unit*.

– *Scaling criteria*: it is the policy that determines the voltage/frequency level of the next scaling point.

Depending on scaling points, DVFS policies can be classified as interval-based policies (timer interrupts) [16, 17], micro-architecture-based policies (cache misses and performance counters),and task-based policies (task arrivals and completions) [18]. Other techniques have also been introduced for multiprocessor embedded systems, such as [19], focusing mostly on scheduling algorithms.

3 Mapping and Exploration Design Flow

The proposed mapping flow allows the designer to co-explore the application design space including both the architecture model and the application source code, as shown in Figure 1. This flow is targeted to highly parallel applications, like, for instance, multimedia ones. This flow starts from an executable specification of an application, in a

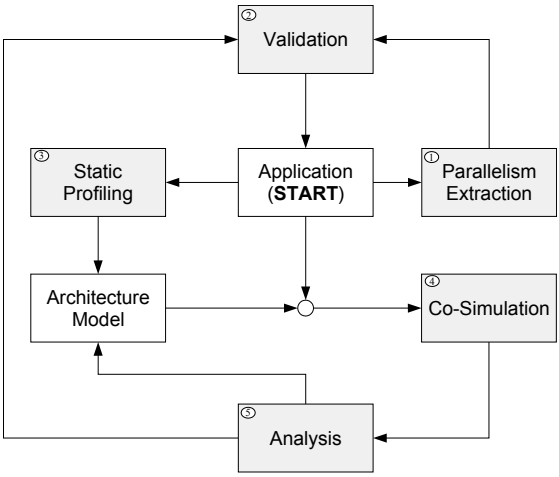

Fig. 1. The proposed mapping and exploration flow

language that can be compiled and executed on the target micro-architecture platform. The mapping phase starts by considering a fully software implementation. The application undergoes the following steps:

– *Static Profiling*: the source code of the application is profiled natively on a workstation and the total computation effort needed by the application is estimated, given performance constraints. It is worth noting that this provides the *lower bound* to the number of processors, in case of a fully software solution.
– *Parallelism Extraction*: since the target architecture is an MPSoC, the intrinsic parallelism must be extracted from the application to exploit the available system resources. The general problem of automatic parallelization of the code is out of

the scope of this paper. We consider code that has been already parallelized for the MultiFlex programming models. This step identifies data dependencies at very coarse-grain, e.g. image macroblocks for an MPEG4 encoding algorithm.
- *Validation*: any modification to the reference source code is validated against the reference data, to avoid losing the original program behavior due to some neglected dependencies.
- *Architecture Modeling*: static profiling provides the lower bound in terms of computational resources to run the application and to meet the constraints, hence we need to model the architecture components considering these basic requirements. As an example, the static profiling states the minimum number of processors for a fully software implementation.
- *Co-simulation*: the architecture model and the software are co-simulated, extracting both performance and power measures.
- *Analysis*: simulation results are collected and analyzed to modify both the application (e.g. to increase the parallelism) and the architecture model, by modifying the initial hardware and software partitioning, for example by moving some parts of the application in hardware or by varying some platform architectural parameters.

The mapping continues to cycle through these steps until the constraints for the application are fully met. These steps are detailed in the following.

3.1 Static Profiling

Statically profiling an application consists of determining its computational requirements. The idea is to define a *lower bound* to the resources needed by the application. In this way, it is possible to configure the micro-architecture platform so that it satisfies the lower bound. Static profiling can be done natively on any machine that supports a profiling tool-set, like `gprof`or `iprof`. `iprof` identifies the number of instructions needed to execute the native code, while `gprof` provides analysis on the call graph. Merging the two outputs allows the designer to identify the computational needs of the application and how those need are distributed in the code. As an example, an MPEG4 encoder developed at STMicroelectronics (described in detail in Section 5), when profiled on a x86 architecture, shows a requirement of 4.081 GIPS (Giga Instructions Per Second). It is worth noting that this requirement is strictly valid only on the same Instruction Set Architecture (ISA) on which the application was profiled. More complex (or simpler) ISAs may run the code using less (or more) instructions.

Considering a micro-architecture platform where the target processor is an ARM9 core running at 200MHz, the bound for a fully software implementation is 21 ARM CPUs. In fact, ARM9 has at most a CPI (cycles-per-instruction) equal to 1:

$$\begin{aligned} c \times f \times CPI &= 21 \times 200 \cdot 10^6 \times 1 \\ &= 4200 GIPS \end{aligned}$$

with c and f the number of processors and their frequency respectively. Even supposing that the micro-architecture platform has enough computational power, a sequential application cannot run on all the processing elements and therefore the designer needs to modify it to exploit all possible inherent parallelism.

3.2 Parallelization

Parallelization requires the identification of sections of code working on independent sections of the application's input data. This is not trivial even for applications that exhibit an "embarrassing" level of parallelism like multimedia audio and video encoding. The kind of parallelization needed for distributing tasks to the processing elements of an MPSoC is coarse- to medium-grained. Although there is a large amount of research devoted to the automatic parallelization of code [20, 21], coarse-grain parallelizing compilers are still not widespread in the industry. It is more common to directly apply programming models to sequential code, leaving the designer to identify data-dependencies in the code. As an example, a routing application like an IPv4 forwarder, can be parallelized in such a way that every packet is manipulated by a separate thread. The use of the MultiFlex SMP approach reduces the effort [2] because it provides a well-established parallel API, that is entirely similar to multi-threaded programming when using a traditional UNIX-like operating system, in particular POSIX threading. In addition, the Concurrency Engine (a hardware SMP accelerator that is part of the MultiFlex approach) provides load balancing functionalities, leaving the designer to the only effort of determining the data dependencies of the target application. The proposed flow

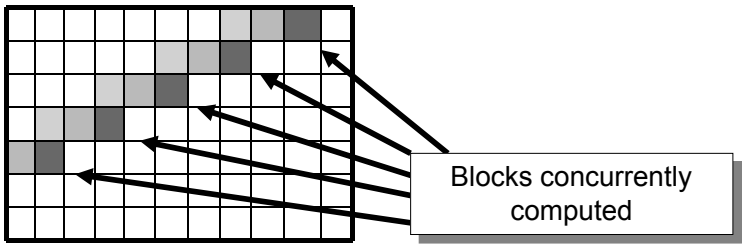

Fig. 2. Parallelization of a motion estimation algorithm over the macroblocks of a frame

involves the application of the MultiFlex SMP programming model, identifying data dependencies manually. As an example, considering STM's MPEG4 encoder, independent data blocks in the motion estimation algorithm are represented by a "knight's move" on the chessboard formed by dividing a frame to be encoded in macroblocks (squares of 2x2 blocks of 8x8 pixels), as typical in JPEG and MPEG compression [22]. The parallelized application starts a thread for each independent macroblock to be encoded, as shown in Figure 2.

3.3 Validation

After some parts of the application have been parallelized, the resulting code has to be verified again in order to prove that the functional requirements of the application still hold. Applying regression tests to simulated code can be excessively time consuming and may seriously slow down the design process. To overcome such problem, we exploit the similarities of the MultiFlex programming model to Pthreads. It is conceivable

to compile natively the same code that would run on the micro-architecture platform, given that the API is the same. Using Pthreads it is possible to explore the parallelization of the application natively, obtaining the exact same behavior that would take place on the simulated platform. The only attention to be given is to use only those Pthread's concurrency control structures that have been implemented in the Concurrency Engine (such as semaphores, monitors, conditions). Validating the parallel design implies enforcing proper synchronization through the use of those structures avoiding any sort of deadlock or race condition. The time saving introduced by this solution is conspicuous. As an example, a simulation run over 30 frames of the MPEG4 encoder takes about 1s natively, while it requires minutes (or even hours, depending on the accuracy level) when it is executed, on the same workstation, with an instruction-set simulator (ISS).

3.4 Simulation

Once the first run of mapping has been applied, the architecture model and the software can be co-simulated. Co-simulation is performed using a transaction-level simulator (in our case, StepNP [5]), first using timed functional models. The first run of simulation provides data concerning the actual performance of the system, giving bounds to delay and energy consumption. The results of the simulation provide data to proceed in the exploration of the platform configuration. Examples of these results might be low processor utilization, which mean that channel latency is excessive and has to be accounted for, as an example using hardware multi-threading [23]. Whenever the application meets the required constraints, simulation can be performed at a lower abstraction level, going into progressive refinements that lead to the actual implementation of the system.

It is unlikely for a complex, high speed, application, that a full software solution gives acceptable performance. In this case, some parts of the system may be implemented in hardware, raising the performance but, at the same time, raising the platform's design cost and lowering its flexibility.

3.5 Hardware-software partitioning

One of the main advantages of the proposed flow is the ease of HW/SW partitioning through the use of DSOC. In fact DSOC does not make a distinction, from the point of view of the user, between hardware and software components. The DSOC ORB routes requests and the MP engine takes care of the marshaling and unmarshaling activities. In fact, it is possible to create transaction-level models of the application functions using the exact same code that is used for software models, as shown in Figure 3.

During the static profiling phase, the critical kernels of the applications have been identified. These are implemented as DSOC objects, defining their function signatures as appropriate interfaces using SIDL. The SIDL compiler generates skeletons and stubs needed for communication between DSOC clients and servers. Using a transaction-level DSOC object adapter, it is possible to connect any DSOC object to a communication channel. Since the code used by DSOC objects can be either compiled for the simulated processors or natively, it is kept into a separate library compiled in both forms. Being StepNP (and MP4Free) based on SystemC, which produces native executable

simulators, the natively-compiled library can be linked directly to the simulator, and accessed through the stubs. This allows the creation of untimed functional models of each component modeled as DSOC. Models can be turned into timed functional ones adding appropriate `wait()` statements in the SystemC wrapper. It is worth noting how this approach is not specific to a type of function or hardware component, but it is completely general: it is sufficient to define a SIDL interface to connect a new object.

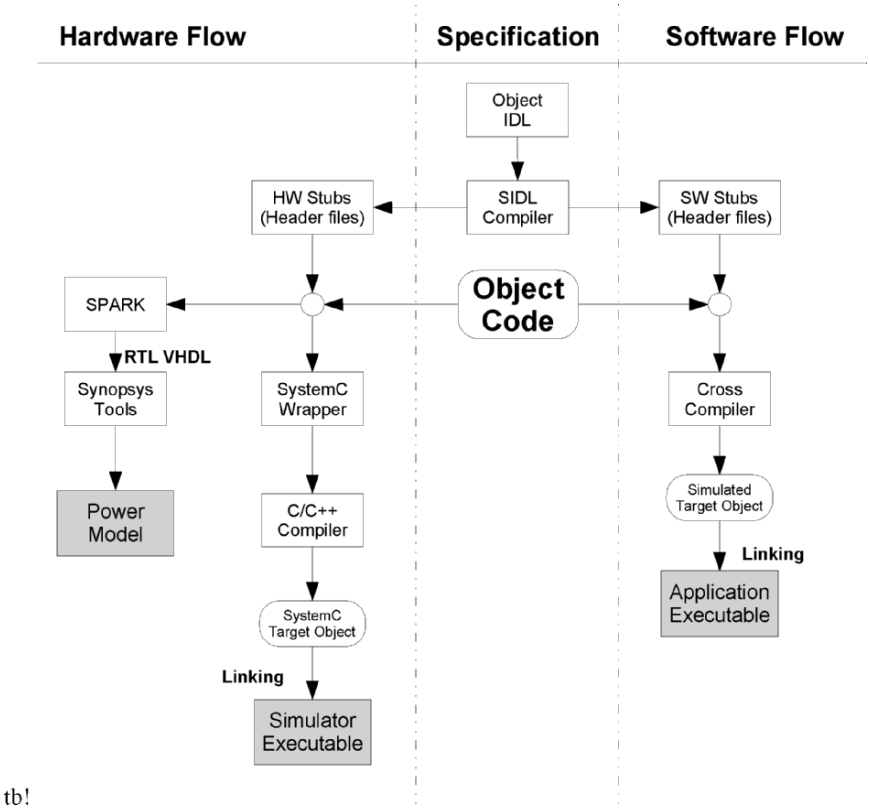

Fig. 3. Exploiting DSOC to re-use code for both hardware and software object models

The proposed methodology, however, suffers from a minor limitation. Since the simulated application and SystemC have different address spaces, it is not possible to use pointers when passing parameters to hardware components. This means that is is not possible for a hardware component to access memory via DMA, unless the SystemC wrapper is sophisticated enough to support address space conversion.

Concerning power consumption, the timed functional models associated with each DSOC object have no power model. To overcome this limitation, we roughly estimated the energy consumption per access to the components using SPARK [24]. SPARK is a behavioral C synthesizer, that produces RT VHDL code. The VHDL code is synthesized

as well in Synopsys Design Analyzer and estimated with Power Compiler. The result gives a first estimation of the energy cost per access to the component, as shown in Figure 3.

Exploiting DSOC, SystemC and TLM, the designer can perform the HW/SW partitioning of the system as a matter of turning some switches and running simulations, greatly simplifying high-level design space exploration. The methodology has been applied to a set of critical kernels, as outlined in the following.

4 Power-Aware Scheduling on MPSoCs

After the application has been partitioned and mapped to the MPSoC platform, in this work we also propose a power-aware task-driven scheduling algorithm that, with respect to previous approaches, works at run-time without the need of an operating system on a multi-processor system. The basic idea is to exploit the MultiFlex [2] hardware-assisted programming models and task structures to gather the necessary information for the scaling criteria to assign voltages to processors. The scheduler is included in the Concurrency Engine as a hardware component, and, if only hardware threads are used, does not require any operating system support except that needed to access the Concurrency Engine (CE). The scheduler acts transparently, identifying task events (start, finish, resource availability) snooping the requests to the CE core. The proposed voltage scheduler, described in the following, has three key advantages:

1. It is a task-driven scheduler but nevertheless does not require any operating system support.
2. It is based on very simple, constant time algorithms
3. Its granularity is very coarse, trying to get as close as possible to the optimal value of one voltage setting per deadline [25]

Due to the structure of most common SMP programming models, i.e. a main thread forking parallel worker threads, it is easier than in the general case to predict the next value of the average load of a processor for the next scheduling unit, due to the predictable behavior of the system. The scheduling points are the fork and join operations (therefore the classification of the policy as event-driven) defined by a predictive approach, outlined in the following. We define the *main thread* the hardware thread executing the main flow of a program, and *worker thread* every other thread. *Tasks* are functions or programs executed by worker threads; *jobs* are instances of a task, that is, they are a mapping between a task and a working data set. A fork operation consists of creating a set of jobs that have to be executed by worker threads. According to the MultiFlex model, only hardware threads are allowed in the system, and the CE maps jobs to threads according to its own internal scheduling algorithm.

For each scheduling unit, we determine the best load obtained for each processor, then use them to compute the predicted value of the next scheduling point. As a prediction scheme, we use an exponentially smoothed moving average, obtained as a weighted average of the best loads of the given task. This prediction scheme works well for applications whose tasks roughly keep the same behavior in time. The assumption is not a limitation if tasks (i.e. functions) don't have a timing behavior that is strongly

dependent on data. Nevertheless, it is possible to use another and perhaps more effective prediction scheme. The average load of a processor is determined with its internal instruction counter and is a number less than or equal to one. The idea is to scale the voltage/frequency of all the processors to keep the average load as close as possible to one. Load average tends to be lower than one due to wait states caused by channel latency, contention, etc..

The targets of the MultiFlex approach are multimedia and network applications: these applications require a huge amount of computational resources, but they are also quite linear and repetitive. This linearity allows collecting statistics on the current program in order to manage future iterations of the same task. According to the programming model, the sequence of fork-join of an application is computed sequentially, and tasks are forked to one or more threads. The main thread is stalled until all the worker threads have completed their task. For each task we keep track of the worst execution time τ as an exponentially smoothed moving average. At every scheduling point, the voltage/frequency of the processor is set according to the predicted worst load average for the next set of jobs. Since most DVFS processors can modify their voltage settings in discrete intervals, we approximate the setting to the voltage/frequency tuple $s_i = (V_{dd}, f)$ such that

$$f \geq f_{max} \cdot l_w \tag{1}$$

where f_{max} is the maximum frequency value and l_w is the predicted average load for the next scheduling unit. This means that the frequency will be higher or equal to the frequency needed to complete the task without incurring in performance penalties, according to the predicted load. The higher the accuracy of the prediction, the better the DVFS result. This approach can be extended to perform static voltage scheduling, if the execution time of each task is known a priori. Finally, we can assume that the scheduling points are far enough in time to reduce the scheduling points in such a way that the transition cost is not affecting the overall power consumption of the application. This has been proved valid for an MPEG4 encoder application: fork-join sections generate a high number (up to 10^3) of threads, and each task requires a time $t >> t_{tran}$, making the overhead negligible.

With the classic SMP approach, scaling voltage of active processors may still incur in energy waste due to idle processing elements. This may happen in two conditions:

1. The inherent parallelism of the application for a task does not allow forking enough threads to cover all the processing elements of the target platform. This means that some processors will be idle during the execution, until the next join.
2. Job distribution is not uniform during a join phase: some jobs finish earlier than others and leave their processor idle.

To avoid this energy expenditure, the DVFS subsystem of the CE applies DPM methodologies, turning off the unused hardware if the conditions arise and this does not affect the performance of the system due to restart delays.

In the following, for the sake of simplicity, we will consider mono-threaded processors, i.e. processors that can execute only one thread at a time. However, the approach still holds for multi-threaded processing elements: a processor executing n threads is considered as n single-threaded processors that belong to the same voltage cluster and can be turned off if and only if all n threads are idle.

4.1 The Concurrency Engine Scheduler

To maximize the effectiveness of the approach, the DVFS subsystem of the Concurrency Engine has to interface with its multi-processor task scheduler. Our approach is designed to interface with both static (fixed for all the tasks) and dynamic scheduling approaches, but in this work, we will focus on the standard dynamic scheduler of the Concurrency Engine. The original implementation of the CE scheduler is a simple FIFO: every new job is mapped in sequence to available worker threads. Every time a job is completed, if there are some unmapped jobs, the first in the queue is assigned to the newly freed resource.

To manage the state of the processors (idle, on, off), the CE monitors the execution of every job. Whenever a task is about to be completed, some processors become idle, but turning them off immediately is not necessarily the most effective strategy. In fact, it is possible that their restart is too slow for the next job to be scheduled without delays, hindering the system's global performance. In the original CE implementation, after the last job is assigned, all the processors are active, finishing their jobs before deadline τ, given by the application constraints (e.g. 30 fps for a video encoder). At the time τ all processors are ready for the next fork, but in the interval between the completion of their job and τ, they are idle. We define the *critical time* $t_{c,i}$ of a task i, the minimum time required to turn off and restart a processor before the beginning of the next task. Finally, we define the *shutdown time* t_d and the *start time* t_s as the time needed to turn off a processor and to turn on one that was previously off, respectively. The time $t_{s,i}$ is the time during task i after which it is not possible to turn on a processor before the next task.

4.2 Scheduling with on/off management

Since the CE possesses all the information needed to schedule tasks over free resources, we can add the proper signals to switch on and off each processor independently. Each processor needs a non-null time to change its state (in either direction), so the CE must check if there is enough time available to shutdown and restart a processor before the beginning of the next task, considering also the voltage schedule of the current task.

Since the DVFS subsystem predicts the duration of a task τ, the CE has an estimation of the time needed to execute the task. Supposing that the restart time is known as $t_{rs} = t_d + t_s$, where t_d is the shutdown time and t_s is the start time. Therefore:

$$t_{c,i} = \tau - t_{rs} \tag{2}$$

The improvement when compared to the standard CE scheduler is little: only processors that can be restarted before τ are turned off, that is processors are stopped only if the number of remaining jobs is lower than the number of available processors, and there is no guarantee that there will be any in all tasks. The constraint that all the processors have to be active at time τ, before starting a new job assignment, can be relaxed if not all the processors are needed for the next task. This can be discovered by the CE after a first conservative run where all tasks are supposed to be needing all computational resources, or can alternatively be defined at design time. In this case, shutdown

and restart operations can be allowed over task boundaries (i.e. when there are no more waiting jobs to be mapped), leading to three cases:

– Case 1: $\tau_{i+1} - \tau_i > t_{rs}$
  In this case it is possible to shutdown a processor in any moment between t_0 and τ_0. Considering the example in Figure 4, supposing that the task ending in τ_2 needs only 3 resources, processors $P2$ and $P5$ can be shut down and their restart can be scheduled at $t_{c,2}$ without incurring in any penalty. This case provides the maximum

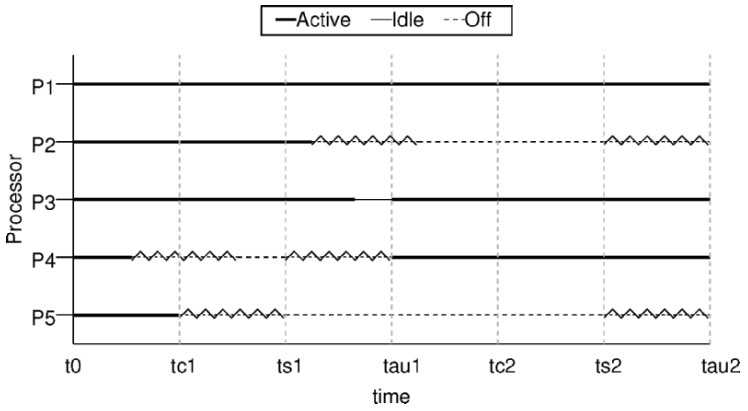

Fig. 4. Shutdown over task boundaries with $\tau_{i+1} - \tau_i > t_{rs}$

  power saving, because it enforces loose constraints on the processors' shutdown process.
– Case 2: $t_{start} < \tau_{i+1} - \tau_i < t_{rs}$
  In this case, it is only possible to start (and not restart) a processor before the next deadline
– Case 3: $\tau_{i+1} - \tau_i < t_s$
  In this case, it is not possible to stop and restart any processor before τ_2

Scheduling with task reordering It is also possible to reorder job mapping to processors so that at the task boundary, the number of required resources k_{next} is less than the number of processing elements k. In fact, the programming model implies that each job is independent from the others, as each job is working on a different data set. Roughly, each job will take the same execution time σ when executed repeatedly using the same resources. If there are k processing elements and there are n jobs to be completed, there are always k active jobs until the last job is scheduled, and the total execution time is:

$$\tau \approx (n \bmod k) \times \sigma \tag{3}$$

We call each set of k jobs an *iteration* of the task. If the number of jobs n is a multiple of the number of processors, then it is not possible to perform reordering among itera-

tions. However, if $k \; mod \; n \neq 0$, it is conceivable to reorder the jobs mapping among iterations, to reduce the fragmentation of idle times at task boundaries.

Given the number of processors k and the number of jobs n, supposing initially, that $n > k$, the number of mapped jobs for all the iterations of the task is described by the regular expression: $[k] + \lambda$, where

$$\lambda = \begin{cases} k & \text{if } n \; mod \; k = 0 \\ n \; mod \; k & \text{otherwise} \end{cases}$$

As an example consider $n = 16, k = 5, \lambda = 1$, in this case $[k]+\lambda =$ "5551". Therefore, the job distribution over the iterations is the sequence 5-5-5-1, 5 jobs for the first 3 iterations and 1 for the last. Let us rewrite the expression as $k'k^*\lambda$, where $k' \in [1, k]$. Keeping constant $k' + \lambda$, it is possible to rearrange the job assignment to best fit the scheduling of shutdown and restart operations in the first and the last iteration. We define the tuple (k'_i, λ_j) the value of λ and k' for the ith and jth task, respectively.

As an example, assume that $k = 5$, then $\lambda = \{1, 2, 3, 4, 5\}$, and $k' = 5$; with these values, all the possible pairs of (k', λ) and their rearrangements are: $(5, 1) \Rightarrow (3, 3), (5, 2) \Rightarrow (4, 3), (5, 3) \Rightarrow (5, 3), (5, 4) \Rightarrow (5, 4), (5, 5) \Rightarrow (5, 5)$.

This arrangement reduces the probability that $\lambda_j > k'_{j+1}$. In fact, if $\lambda_j > k'_{j+1}$, some processor may be idle at the beginning of the first iteration of task $j + 1$. Considering the example and two consecutive tasks, there are 9 different (λ_j, k'_{j+1}) tuples, and only 3 have $\lambda_j > k'_{j+1}$. In the worst case, 2 processors every 4 join operations will be in idle state for a time smaller than t_{rs}. Removing the hypothesis that $n > k$, in the worst case, a processor can be in an idle state for an iteration. This reordering table is computed at design time and hard-coded in the CE, and depends on the number of processing elements available in the target SoC. The CE FIFO scheduler uses the reordering table for scheduling whenever it receives a fork command by the main thread.

If we consider that, in general, the execution time needed by a task of more than a few assembly instructions is greater than t_{rs}, it is possible to merge the three cases presented in Section 4.2 into a single algorithm. The algorithm merges the last and the first iteration of two consecutive tasks, considering two separate deadlines. The behavior is shown in Figure 5, and the algorithm follows:

```
1:  scheduled_active = 0;
2:  for all P_j do
3:     if P_j completes his job before t_{c,i} then
4:        shutdown P_j;
5:        if length(scheduled_active) ¡ k_{next} then
6:           schedule P_j for restart at t_{s,i}
7:           scheduled_active++;
8:        else
9:           schedule P_j for restart at t_{s,i+1}
10:       end if
11:    else if P_j completes his job in [t_{c,i}, t_{c,i+1}] ∧ k_a ≥ k_{next} then
12:       shutdown P_j;
13:       schedule P_j for restart at t_{s,i+1};
14:    else
```

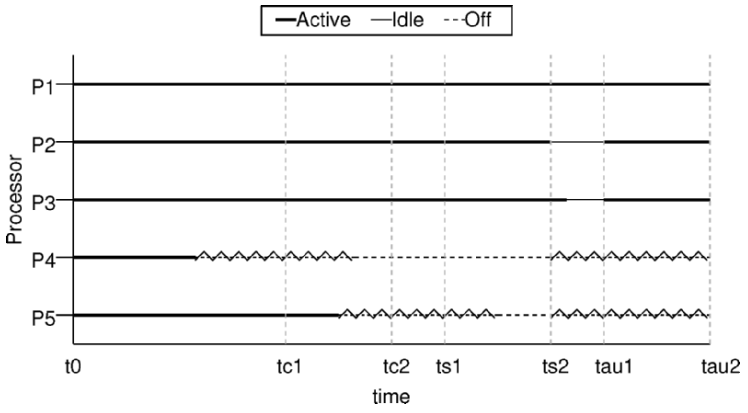

Fig. 5. Behavior of the generalized DPMS

15: scheduled_active++;
16: **end if**
17: **end for**

It worth noting that if $\tau_{i+1} - \tau i > t_{rs}$, we obtain Case 1, and if $t_s < \tau_{i+1} - \tau i < t_{rs}$, the result is an improved arrangement of Case 2, that were detailed in Section 4.2. Theoretically, it is possible that $\tau_i + \tau_{i+1}$ is lower than the time required to restart a processor and only in such case, the scheduler considers also τ_{i+2} and so on. However, it is reasonable to assume that the time required to execute a task is significantly larger than the time needed to restart a processor, and therefore only two deadlines have to be considered when scheduling shutdowns and restarts. In fact, considering as an example an MPEG4 algorithm parallelized for the purpose, each task requires on average $7ms$ while an ARM10 processor running at 200MHz requires $\sim 10\mu s$to awake from shutdown mode.

The introduced algorithms have linear complexity in the number of computational resources, making them viable for a fast hardware implementation.

5 Case Study: an MPEG4 encoder

This section describes how the methodology was applied to map an MPEG4 onto the MultiFlex platform. This platform is constituted of a variable number of ARM processors with a variable number of hardware threads, a two-level cache structure and the STBus interconnection network [12].

The application, specified in C, has been initially profiled statically, using gprof and iprof (GNU open source tools) on a Linux machine. Profiling defined a lower bound on the number of processors needed for execution: the computing power needed by the applications amounts to 4.08 GIPS. A full software solution would require a minimum of 21 ARM CPUs running at 200MHz (each one providing at most 200MIPS). Table 2 shows the results for the 9 functions that take most of the execution time during the encoding of a frame. These functions represent only a very small portion of the

application code (approximately 6% of 8086 lines of code) but they cover 82.81% of all computational resources needed for execution.

Table 1. Static profiling results of the MPEG4 application on Linux

Function	Execution Time [%]	Lines of Code	Fraction of source [%]
BSAD	27.98	90	1.11
BQ	19.17	100	1.21
BDCT	10.36	80	1.11
BZIGZAG	6.22	5	0.06
BIDCT	5.70	110	1.2
BADD	4.66	15	0.18
BDIFF	3.63	17	0.21
BQI	2.59	37	0.45
BSUM	2.59	10	0.12
TOTAL	82.81	465	5.65

As a first design choice, these blocks were selected for a possible hardware implementation in a coprocessor or MPEG4 accelerator: these functions are present in all versions of the MPEG algorithm [22], and moving them to hardware blocks does not hinder the overall flexibility of the system. These blocks correspond to 83% of all computation time but less than 6% of all the application lines of code. These functions were modeled as DSOC servers, with the application software accessing either the hardware or software versions of the models.

The remaining 17% of the application computation is executed as software. The profiling of the distributed application shows that 800 MIPS are required to run the application on the ARM processors. The data access bandwidth of these processors is 1.7 GB/s.

Concerning the hw/sw partitioning of the application, we applied DSOC programming model allowing to easily switching from software to hardware an vice-versa, using timed functional models. The code was compiled and executed on StepNP in an Instruction Set Simulator (ISS) when simulated as software and it was instrumented for timing analysis, compiled natively, and executed in SystemC space when simulated as hardware.

Adding power estimation required to build models for the hardware components. To simplify the modeling, we used SPARK [24] and synthesized RTL directly from the source code, and we used Synopsys Power Compiler to derive a power model based on the access to the devices using STM technology libraries at 0.18μm, as shown in Table 2. The use of SPARK provides good results since the MPEG4 high-computation kernels are very simple functions.

Switching progressively each kernel to hardware, starting from a full-software solution and from the most computationally-intensive kernel, brings to the results shown in Figure 6. It is remarkable how turning BDCT into hardware constitutes a significant

Table 2. SPARK results for the MPEG4 critical kernels

Function	Cell Pow. [μW]	Interconnect Pow. [μW]	Total [μW]	Leakage [μW]
BSAD	17,028	16,063	33,091	20,494
BQ	6,426	1,572	7,999	5,903
BDCT	0,156	0,009	0,165	0,060
BZIGZAG	5,133	1,716	6,850	4,837
BIDCT	23,179	14,037	37,216	24,761
BADD	16,069	5,247	21,316	15,170
BDIFF	38,160	14,114	52,274	29,481
BQI	4,308	1,430	5,739	7,057
BSUM	0,329	0,781	1,110	3,035

energy saving, while it does not have the same effect on performance. Only turning both BDCT and IDCT into hardware has the effect of raising the performance. This means that the DCT is one of the major bottlenecks of the system, and therefore justifying the hardware design choice. The frame rate increases as more components are turned to hardware, but this is not sufficient to reach the 30 frames per second constraint, and more optimizations have to be done in terms of parallelism exploitation.

To exploit the MPSoC architecture, the MPEG4 application has then been split into parallel sections working on independent data. This phase has been optimized manually for the target MPEG4 application. The inner loops were parallelized using fork-join constructs. Data dependencies were carefully analyzed and verified after parallelization.

The architecture has been explored with the proposed flow, and the graph of Figure 7 summarizes the overall performance results, expressed in frames per second (fps) achieved, for a range of architecture parameters. These include the number of processors (2 to 5) and the number of threads per processor (2 to 8). The upper curve represents

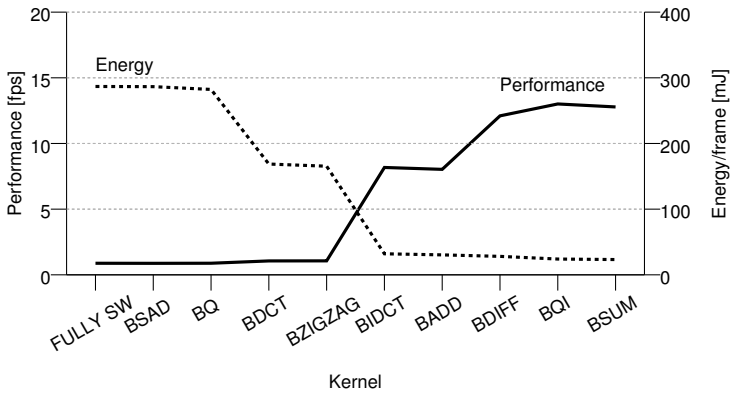

Fig. 6. Performance and power efficiency of the MPSoC with varying hardware/software partitioning

the theoretical upper bound for a perfect parallelization (i.e. results for a single processor accessing local memory, and then simply multiplied by the number of processors). This theoretical result does not include any inter-processor communication code and assumes zero bus latency. The best result makes use of 5 processors and 8 hardware threads per processor. In this case, 28.5 fps is achieved, or 86% of the theoretical best result of 33 fps. The system was simulated with and increasing number of processors, after parallelization, showing a result close to the theoretical upper bound when using a no-wait-state channel and a result very close to the latter when using STBus.

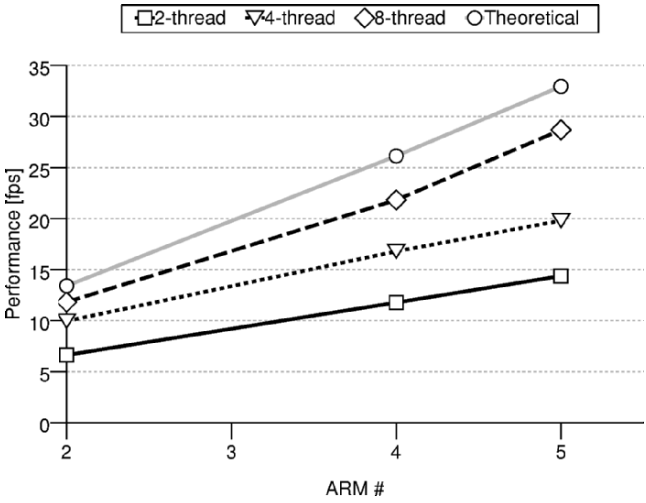

Fig. 7. Performance of the MPEG4 application with different configurations

5.1 Power optimization

Application mapping results are very effective for the given platform, as the overall average load for each processor is roughly 85%. Therefore, we cannot expect large savings in power consumption, because the available resource usage is only 15% away from maximum usage. Nevertheless, it is still possible to gain a 10% power saving applying the methodologies outlined in this work. For less efficient mappings, these savings might be even larger.

The DVFS scheduling algorithm was tested on the platform and the resulting voltage schedule is shown in Figure 8. The algorithm managed to increase the average load per processor from 85% to 96%, and after the first three frames (during which the MPEG4 pipeline was filling up) the voltage stabilizes to the best feasible value without affecting the algorithm performance, maintaining the average load to its best value.

Concerning the processor state management, the results are shown in Figure 9: the number of cycles spent by the processors in idle state (but fully powered) is reduced by

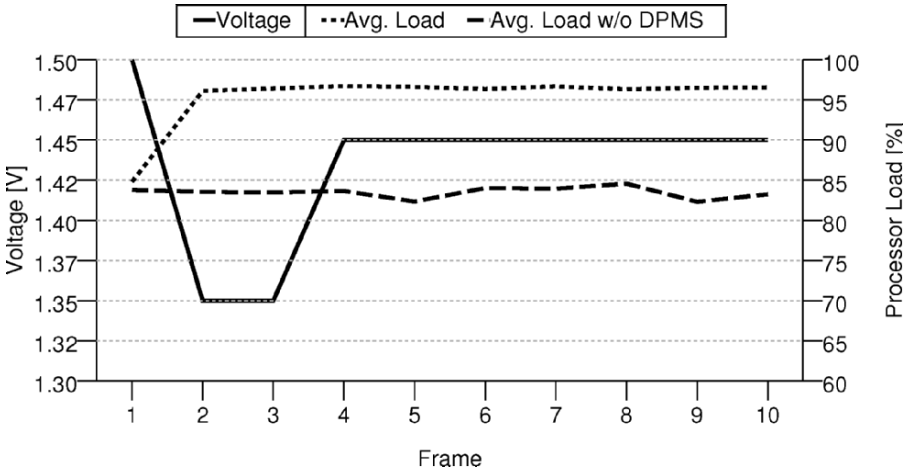

Fig. 8. Voltage scheduling performed by the algorithm and energy results

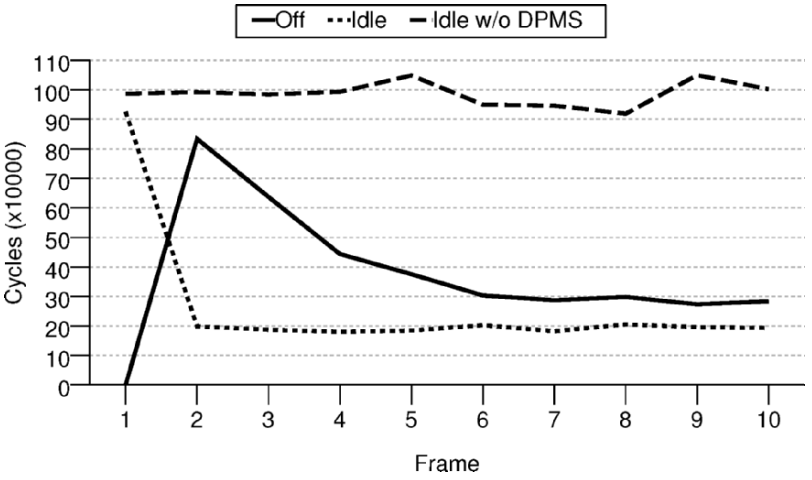

Fig. 9. Average cycles in different states of the processors

80%, and processors spend more time in low-power mode (the Off state in the figure). The remainder number of idle cycles is spent while switching from one state to the other.

Due to the high efficiency of the mapping an the state swticthing cost for the arm processor, the overall energy saving is roughly 10%, which is inline with the increase in average load per processor.

6 Concluding Remarks

This paper presented a mapping methodology for applications on the MultiFlex platform. In addition, this work presents a fast partitioning exploration scheme that takes advantage of the DSOC programming model. These methodologies have been applied to a multimedia case study: an industrial MPEG4 encoder, showing the validity of the approach. Future works include the integration of the methodology with automatic exploration algorithms and automatic parallelization of the application code. Concerning power consumption, this work also introduces a novel low-power voltage scheduler and a dynamic power management system for the MultiFlex system. This DPMS has three key advantages: it is task-driven without needing any operating system support, its algorithms are linear in the number of computational resources, and the scheduling granularity is very coarse compared to the target application structure. Future work will add multiple sleep states for the processor cores (with different wake-up times) and compare scheduling results with optimal values.

References

1. Keutzer, K., Newton, A., Rabaey, J., Sangiovanni-Vincentelli, A.: System-level design: Orthogonalization of concerns and platform-based design. IEEE Transactions On Computer-Aided Design of Integrated Circuits and Systems **19** (2000) 1523–1543
2. Paulin, P.G., Pilkington, C., Langevin, M., Bensoudane, E., Nicolescu, G.: Parallel programming models for a multi-processor SoC platform applied to high-speed traffic management. In: CODES+ISSS'04: Proceedings of the Conference. (2004) 48–53
3. Cai, L., Gajski, D.: Transaction level modeling: an overview. In: CODES+ISSS'03: Proceedings of the Conference. (2003) 19–24
4. Beltrame, G., Sciuto, D., Silvano, C., Paulin, P., Bensoudane, E.: An application mapping methodology and case study for multi-processor on-chip architectures. In: VLSI-SoC'06: Proceedings of the Conference. (2006)
5. Paulin, P.G., Pilkington, C., Bensoudane, E.: StepNP: A system-level exploration platform for network processors. IEEE Design and Test of Computers **1** (2002) 2–11
6. Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Passerone, C., Sangiovanni-Vincentelli, A.: Metropolis: An integrated electronic system design environment. IEEE Computer **34** (2003) 45–52
7. Dziri, M.A., Cesrio, W., Wagner, F.R., Jerraya, A.A.: Unified component integration flow for multi-processor soc design and validation. In: DATE'04: Proceeding of the Conference. (2004)
8. Jersak, M., Henia, R., Ernst, R.: Context-aware performance analysis for efficient embedded system design. In: DATE'04: Proceedings of the Conference. (2004)

9. Pop, T., Eles, P., Peng, Z.: Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In: CODES'02: Proceedings of the Symposium. (2002) 187–192
10. Pimentel, A.D., Lieverse, P., van der Wolf, P., Hertzberger, L., Deprettere, E.F.: Exploring embedded-systems architectures with Artemis. IEEE Computer **34**(11) (2001) 57–63
11. Kempf, T., Doerper, M., Leupers, R., Ascheid, G., Meyr, H., Kogel, T., Vanthournout, B.: A modular simulation framework for spatial and temporal task mapping onto multi-processor soc platforms. In: DATE'05: Proceedings of the Conference. (2005) 876–881
12. Paulin, P.G.: Automatic mapping of parallel applications onto multi-processor platforms: a multimedia application. In: Digital System Design, Euromicro Symposium. (2004) 2–4
13. Pazos, N., Maxiaguine, A., Ienne, P., Leblebici, Y.: Parallel modelling paradigm in multimedia applications: Mapping and scheduling onto a multi-processor system-on-chip platform. In: Proceedings of the International Global Signal Processing Conference, Santa Clara, California (2004)
14. Zhai, B., Blaauw, D., Sylvester, D., Flautner, K.: Theoretical and practical limits of dynamic voltage scaling. In: DAC '04: Proceedings of Conference. (2004) 868–873
15. Xie, F., Martonosi, M., Malik, S.: Efficient behavior-driven runtime dynamic voltage scaling policies. In: CODES+ISSS '05: Proceedings of the Conference. (2005) 105–110
16. Lorch, J.R., Smith, A.J.: PACE: A new approach to dynamic voltage scaling. IEEE Transactions on Computers **53** (2004) 856–869
17. Choi, K., Soma, R., Pedram, M.: Fine-grained dynamic voltage and frequency scaling for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times. In: DATE '04: Proceedings of the Conference. (2004)
18. Andrei, A., Schmitz, M., Eles, P., Peng, Z., Al-Hashimi, B.M.: Overhead-conscious voltage selection for dynamic and leakage energy reduction of time-constrained systems. In: DATE '04: Proceedings of the Conference. (2004)
19. Kadayif, I., Kandemir, M., Vijaykrishnan, N., Irwin, M., Kolcu, I.: Exploiting processor workload heterogeneity for reducing energy. In: DATE'04: Proceedings of the Conference. (2004)
20. Bacon, D.F., Graham, S.L., Sharp, O.J.: Compiler transformations for high-performance computing. ACM Computing Surveys **26** (1994) 345–420
21. Banerjee, U., Eigenmann, R., Nicolau, A., Padua, D.A.: Automatic program parallelization. Proceedings of the IEEE **81** (1993) 211–243
22. Murray, D.J., VanRyper, W.: Encyclopedia of Graphics File Formats. O'Reilly Associates (1996)
23. Beltrame, G., Palermo, G., Sciuto, D., Silvano, C.: Plug-in of power models in the StepNP exploration platform: Analysis of power-performance trade-offs. In: CASES'04: Proceedings of the Conference. (2004) 85–92
24. Gupta, S., Dutt, N., Gupta, R., Nicolau, A.: SPARK: A high-level synthesis framework for applying parallelizing compiler transformations. In: VLSID'03: Proceedings of the Conference. (2003)
25. Ishihara, T., Yasuura, H.: Voltage scheduling problem for dynamically variable voltage processors. In: ISLPED'98: Proceedings of the Symposium. (1998) 197–202